# System management for AIPS++

## Report on progress to 1992/May/06

*Mark Calabretta*
**AIPS++** programmer group
1992/May/06

## 1 Introduction

The design of the **AIPS++** system has been described in two previous documents "System management for **AIPS++** - Part 1: organization and distribution", and "Part 2: activation, generation, and verification". A report describing progress to 1992/Mar/14 was delivered to the March **AIPS++** steering committee meeting in Socorro. This paper describes further progress in implementing the design, particularly concerning the code distribution system.

Most of the major functional elements of *AIPS* system management are now present to some extent in **AIPS++**. Although not necessarily a good indicator of progress, the statistics of code written may be of interest:

| | |
|---|---|
| `shell scripts:` | `2778` |
| `aipsinit and aipshosts:` | `354` |
| `aipsrc, aipsrc templates, and getrc:` | `260` |
| `makefiles, makedefs, and makedefs templates:` | `1417` |
| `man pages for the above:` | `676` |
| | `----` |
| `total:` | `5485` |

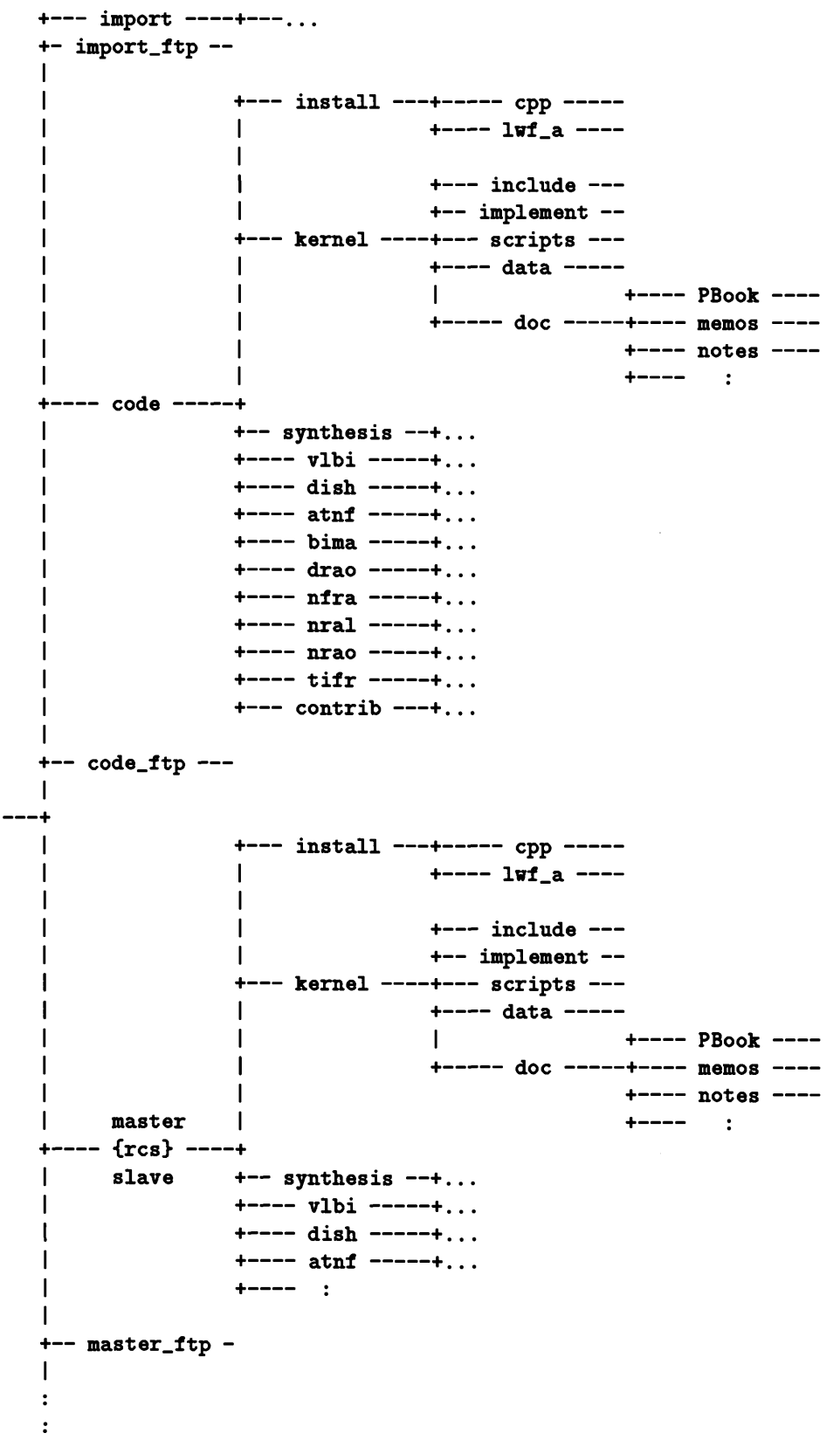
Development will be required throughout the life of **AIPS++**, but I would estimate very subjectively that the system is about 70% complete,

## 2 The directory tree

The **AIPS++** code and system directories have been implemented essentially as described in the original documents. One issue which was not canvassed previously is the location of the *RCS* source code repositories. These have been implemented as a separate and parallel directory hierarchy under `~aips++/master`. At the moment the source code management consists of plain *RCS*, but this structure will readily accomodate *CVS*. Plain-text copies of the code are kept under the `~aips++/code` tree as originally planned.

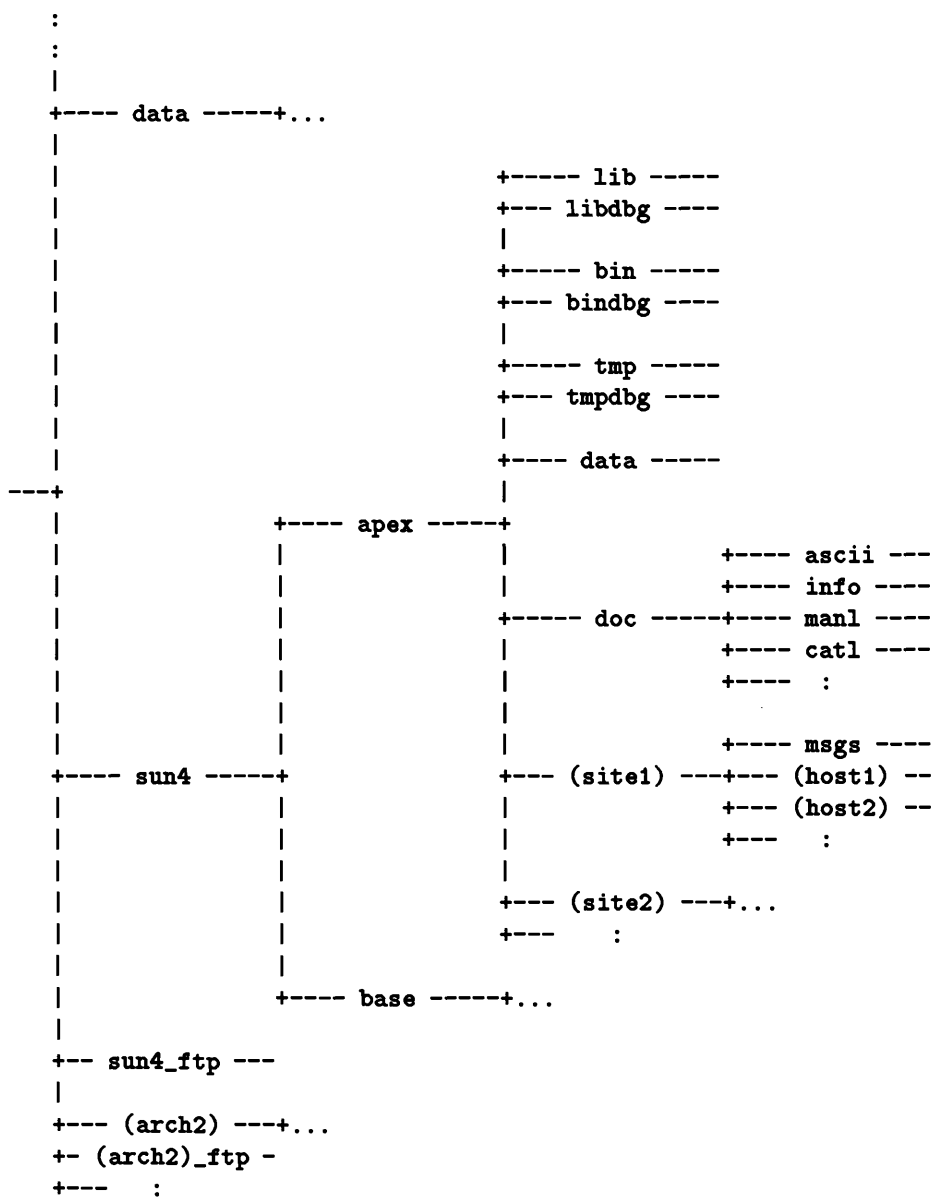
Another point of departure from the original design has been to incorporate the *ftp* directories directly under `~aips++`. This was prompted by the realization that `~aips++` could itself be NFS mounted under `~ftp`, thereby circumventing the *chroot* limitation imposed by anonymous *ftp*. All of **AIPS++** is now available for access via this method.

One modification required to meet this change was that of renaming the *ftp* directories so that they could be accomodated under `~aips++`. Certain other minor changes were also made, particularly with respect to the code distribution files. The structure currently resembles the following:


import
import_ftp
install
cpp
lwf_a
include
implement
kernel
scripts
data
doc
PBook
memos
notes
code
synthesis
vlbi
dish
atnf
bima
drao
nfra
nral
nrao
tifr
contrib
code_ftp
master
{rcs}
slave
master_ftp

A typical **AIPS++** installation would only have the `import` and `code` subdirectories. Consortium sites would also have the `import_ftp`, and `code_ftp` subdirectories for redistribution of **AIPS++** source code to local **AIPS++** recipients. Consortium sites other than Charlottesville will also have a `slave` subdirectory containing a copy of the master *RCS* files. Charlottesville will instead have the `master`, and `master_ftp` subdirectories, the latter of which contains updates for the slave subdirectories at the other consortium sites.

All **AIPS++** installations will have system directories with a structure similar to the following, except that only consortium sites will have `ftp` subdirectories:




This hierarchy is identical to that originally envisaged.

# 3 Distribution

Packaging of the **AIPS++** source code for distribution to end-users, and also of updates for the *RCS* repositories to consortium sites, is handled at Charlottesville by a single script called *exhale*. *exhale*'s behaviour is determined by the contents of, and last modification time of the `VERSION` file in `~aips++/master_ftp`.

If the **VERSION** file contains the string "**mm.-**", where **mm** is a two digit number which defines the major version number, *exhale* interprets this as a signal to produce a new base release. It produces a compressed tar file for the **~aips++/master** tree, and also for each of the **~aips++/code** subdirectories. If all goes well it then installs these in the **~aips++/master_ftp** and **~aips++/code_ftp** directories. If this succeeds, *exhale* finishes by installing a new **VERSION** file containing the major and minor version number with a timestamp and file modification time pertaining to when *exhale* started execution.

If, on the other hand, *exhale* finds a string of the form "**mm.nnn**" in the **VERSION** file, where **nnn** is the minor version number, it will produce an update for the **~aips++/slave** repositories at the consortium sites. Incremental updates consist of a compressed tar file containing all files in the **~aips++/master** repository which are newer than the **VERSION** file itself. In addition, it produces a cumulative update which consists of all **master** files modified since the base release for the current major version. Only after successfully installing these in **~aips++/master_ftp** does it update the **VERSION** file.

The **master** guarantees to create two new updates per day for the **slave**, one by 0700 and the other by 1900 Charlottesville time via *cron* jobs which activate *exhale* a half-hour beforehand.

*exhale* maintains a **LOGFILE** which it updates every time it executes. It is designed to be *fail-safe*, and to date it has operated flawlessly for about a month, except on one occasion when it ran out of disk space and did fail safe. The **AIPS++ ftp** directories currently contain the following:

```
                    +--- README
                    +--- VERSION
                    +--- compress-4.0.1.tar
                    +--- cpp-0.00.tar.Z
    +- import_ftp --+--- make-3.62.tar.Z
    |               +--- make_doc-3.62.tar.Z
    |               +--- perl-4.019.tar.Z
    |               +--- rcs-5.6.tar.Z
    |
    |               +--- LOGFILE
    |               +--- README
    |               +--- VERSION
    |               +--- atnf-00.000.tar.Z
    |               +--- bima-00.000.tar.Z
    |               +--- configure
    |               +--- contrib-00.000.tar.Z
    |               +--- dish-00.000.tar.Z
    |               +--- drao-00.000.tar.Z
----+-- code_ftp ---+--- install-00.000.tar.Z
    |               +--- kernel-00.000.tar.Z
    |               +--- nfra-00.000.tar.Z
    |               +--- nral-00.000.tar.Z
    |               +--- nrao-00.000.tar.Z
    |               +--- synthesis-00.000.tar.Z
    |               +--- tifr-00.000.tar.Z
    |               +--- vlbi-00.000.tar.Z
    |
    |               +--- LOGFILE
    |               +--- README
    |               +--- VERSION
    +-- master_ftp -+--- master-00.000.tar.Z
                    +--- master-00.001.tar.Z
                    +--- master-00.002.tar.Z
                    +--- master-00.003.tar.Z
                    +--- master-00.003.ALL.tar.Z
```

At the remote consortium sites a script called *inhale* fetches the latest consortium updates and applies

them. On its first invokation *inhale* constructs the `~aips++/slave` subdirectory and fetches the base file plus cumulative update. Thereafter it maintains a `VERSION` file in `~aips++/slave`, and decides which incremental updates it needs to fetch from Charlottesville on the basis of the version stored therein. It does, however, accept a "*-c*" option which forces it to fetch the cumulative update. This should be done every once in a while at the remote site in order to ensure that the `slave` is faithfully synchronized with the master.

After fetching the compressed tar files *inhale* unloads them in reverse order. It then invokes GNU *make* to recursively rebuild **AIPS++**. *inhale* also maintains `VERSION` files in `~aips++/code` and `~aips++/$ARCH`. There are corresponding `LOGFILE`s for each of the three `VERSION` files used by *inhale*

Like *exhale*, *inhale* is designed to be *fail-safe* and can be invoked at any time. However, it should normally be invoked by *cron* after 1900 hours local time, and not within an hour of the time scheduled for *exhale* to run in Charlottesville. It works directly on `~aips++/slave` so there should never be any danger of it overwriting `~aips++/master`.

# 4 Installation

Installation of **AIPS++** is under the control of a script called *configure*. One of *exhale*'s jobs in creating a new base release is to install the latest version of the *configure* script in `~aips++/code_ftp`. Intending users of **AIPS++** would *ftp* anonymously to their local consortium site and fetch the entire contents of `~ftp/pub/aips++/code_ftp`. In detail, the sequence would resemble the following:

```
yourhost% mkdir ~aips++
yourhost% chgrp aips2mgr ~aips++
yourhost% chmod 775,g+s ~aips++
yourhost% mkdir ~aips++/code
yourhost% cd ~aips++/code
yourhost% ftp baboon.cv.nrao.edu
Connected to baboon.cv.nrao.edu.
220 baboon FTP server (SunOS 4.1) ready.
Name (baboon.cv.nrao.edu:you): ftp
331 Guest login ok, send ident as password.
Password:you@yourhost.there
230 Guest login ok, access restrictions apply.
ftp> cd pub/aips++/code_ftp
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> prompt
Interactive mode off.
ftp> hash
Hash mark printing on (8192 bytes/hash mark).
ftp> mget *
     :
  lots of messages describing what ftp is fetching
     :
ftp> quit
221 Goodbye.
yourhost% chmod 544 configure
yourhost% configure
```

Once *configure* has been activated the installation consists of answering a few questions, most of which have sensible default answers. *configure* first ensures that certain utilities that it needs are available (*compress* and GNU *make*). If they aren't it will fetch them from the consortium site and if necessary, install *compress* in the current directory. It then unloads any compressed tar files present in the current directory.

*configure* next installs the *aipsinit* scripts (see below) and constructs the `~aips++/aipshosts` database via a simple inbuilt editor. Once the host architectures, site-, and host-names are known, *configure* invokes *aipsinit* for itself and then creates the **AIPS++** system directory tree for each architecture. With a `bin` directory now in existence, *configure* reinstalls *compress* if necessary, and likewise GNU *make*.

Before *make* can be invoked, however, the site-specific `makedefs` file must be defined. This contains definitions used by *make* for directories, compiler options and the like. Generic definitions are provided in `~aips++/$ARCH/$VERS/makedefs` but several of these have no sensible default and must be defined on a site-wide basis. If none already exists, *configure* installs a template `makedefs` file in `~aips++/$ARCH/$VERS/$SITE` and invites the installer to make the appropriate entries with a text editor in another window (or by suspending *configure*). When the user is satisfied, *configure* checks to make sure that all definitions have been made, and then repeats the process for the site-specific `aipsrc` file (see below).

At this point the system should have been bootstrapped to a state where a recursive *make* may be used to check-out all code and rebuild the system, and this *configure* now does.

## 5 *aipsinit*

**AIPS++** programmers and users define their **AIPS++** "environment" by means of the `aipsinit.[c]sh` scripts. One or other of the two scripts can be used for Bourne-like shells (*sh*, *bash*, *ksh*), or C-like shells (*csh*, *tcsh*). *aipsinit* defines a single environment variable, `AIPSPATH`, which contains five space-separated character strings describing the root of the **AIPS++** directory tree, the host architecture, the **AIPS++** version, the local site name, and the local host name. This information is fundamental and must be known in order to access the `aipsrc` databases. *aipsinit* also redefines the `PATH` (and `MANPATH`) environment variables, adding the **AIPS++** binary (and man page) directories to it. The user can control the precise position of the **AIPS++** binary directories in `PATH`, or else can accept the default of prepending it (after "." it that was first).

# 6 Source code management

A set of **AIPS++** accounts and groups have been defined to perform the following functions:

- *aips2mgr* (uid=31415, gid=31415): Owns the **AIPS++** directories, and all *RCS* and *ftp* distribution files. *aips2mgr*, and those in the *aips2mgr* group, are *RCS* and *ftp* administrators for **AIPS++**. In particular, they can directly invoke the *rcs* command on the **AIPS++** *RCS* files. *aips2mgr* will have an active login account only on selected machines. *aips2mgr* is responsible for maintaining the `ftp` directories, and in particular, runs the daily *exhale* and *inhale* cron jobs.
- *aips2prg* (uid=31416, gid=31416): A generic programmer account, although in practice this may not often be used. The *aips2prg* group lists those who have permission to check code into **AIPS++**, or check it out with a write lock (anyone can check code out without a write lock).
- *aips2usr* (uid=31417, gid=31417): The *aips2usr* account itself could serve as a generic **AIPS++** usage account if this is so desired. The *aips2usr* group could be used to control access to **AIPS++** executables and **AIPS++** data areas if this degree of control is desired by the **AIPS++** administrator (the alternative being to give world execute and/or write permission).

Remote consortium sites with access to the *internet* can choose to NFS mount the `master` **AIPS++** *RCS* directories on their local development machines. This is most conveniently done via the `/net` *automounter* map. All that is required is for the fully qualified hostname to be entered in the *aips2-consortium* netgroup which has been created at Charlottesville for the purpose. A symbolic link is created at the remote site as

```
~aips++/master -> /net/baboon.cv.nrao.edu/aips2/aips++/master
```

This gives a view into the `master` *RCS* repositories in addition to the local copy in

`~aips++/slave`

which is updated once a day. All **AIPS++** makefiles access the *RCS* repositories via a symbolic link

`~aips++/rcs`

which is set to point to one or the other. At remote sites it normally points to **slave** and in Charlottesville it always points to **master**.

When a library or executable is rebuilt, the local **rcs** directories are normally the ones consulted. However, it is possible to reset the **rcs** "switch" to **master** at a remote site to cause *make* to checkout and/or rebuild **AIPS++** from the **master** repositories. This has been tried at ATNF Epping with encouraging results.

**AIPS++** programmers create their own "shadow" representation of the **AIPS++** code directory tree by using the *mktree* utility which also creates symbolic links into the `~aips++/master` directories. They then appear to have their own private workspace with a window directly into the master *RCS* repositories at Charlottesville. Whenever they check code out of, or into the system from their private work area they will be dealing directly with the *RCS* repositories in Charlottesville.

Source code checkin and checkout is mediated by *ai* and *ao*. *ai* activates its alter-ego, *ai_master*, which resides in the `~aips++/master` directory itself, and runs setuid to 31415 - *aips2mgr*'s uid in Charlottesville. This mechanism allows programmers at remote sites to check code into Charlottesville, which will certainly have a different set of uids and gids to those at their own site. The remote consortium sites will, however, be encouraged to adopt the uids and gids defined at Charlottesville for **AIPS++**. These were deliberately chosen to be large random numbers to minimize the likelihood of conflict with the remote site's set of uids and gids.

Whenever code is checked in to the **master**, *ai* updates the local **slave** copy (if present - and it won't be at Charlottesville), and also the plain text copy in `~aips++/code`. Since the **slave** repositories are owned by the local *aips2mgr*, *ai* itself runs setuid to its uid (hopefully also 31415).

*ai*'s counterpart, *ao*, has a similar alter-ego called *ao_master* which operates in precisely the same way as *ai_master*, thereby allowing code to be checked out of the **master** with a lock on it (which requires write permission on the *RCS* version file).

A third useful utility, *au* runs setuid to the local *aips2mgr* account and updates **slave** (if present) and **code** with the named files from the **master**.

# 7 System generation

POSIX.2 compliance for *make* applies largely to the way makefiles themselves are written, and essentially restricts functionality to the lowest common denominator of all the many varieties of *make*. This means it would be it very difficult to write portable POSIX.2 compliant makefiles for **AIPS++**, so GNU *make* has been adopted instead. GNU *make* is itself POSIX.1 and POSIX.2 compliant, and so satisfies the "POSIX compliance" criteria originally set for **AIPS++**.

**AIPS++** uses a hierarchical system of GNU makefiles. The head resides in `~aips++/code` and recursively invokes all makefiles residing in the subdirectories beneath it. The makefiles have the following features:

- The makefiles "*include*" architecture, site-, and possibly host-dependent **makedefs** files which allow local definitions to override the defaults, for example, architecture-specific libraries, the location of directories, compiler options, etc.
- The makefiles recognize that the **AIPS++** installation may or may not have a copy of the **rcs** repositories. If the repositories are present, *make* automatically updates the plain-text copy if necessary. This should rarely be necessary though since *ai* keeps it up-to-date.
- The makefiles all contain a **makefile** target thereby causing GNU *make* to invoke its clever mechanism which ensures that makefiles remake themselves before attempting to remake anything else.

- Programmer workspaces are supported by means of GNU *make*'s `VPATH` function. If the programmer has a `.h` or `.C` file in the current directory it will be used instead of those in the `code` areas. In this case the **AIPS++** libraries and executables will not be updated. Instead, the preprocessed code, object module, and/or executable will be left in the programmer's own directory.
- The makefiles construct lists of dependency files via GNU *make*'s `wildcard` function. Any `.h` or `.C` file in the relevant `code` directory (or `rcs` directory if present) is automatically included.
- Any subdirectory is recognized as a target. The `allsys` target of the makefile in the subdirectory is invoked. (This does not apply to `include` subdirectories which do not have makefiles but instead rely on the `implement` makefile for targets such as `chkout`).
- The `implement` makefiles know about the `TESTBED` mechanism by which class test programs are embedded within the class implementation files in an "`#ifdef TESTBED`" wrapper. Executables for the class test programs are handled by a pattern rule triggered by appending `TEST` to the class name.
- The diagnostic targets `test_def` and `test_env` can be invoked for informational and debugging purposes.

By allowing programmers to compile code in their own workspace without having to extract related files from the `rcs` repository, the makefiles minimize the number of files that need to be present in the programmer's own workspace. This reduces the possibility that these may be "stale".

The **AIPS++** makefiles use a standard set of target names; references to a directory in the following list refers to the directory in which the makefile exists, or the corresponding **AIPS++** directory:

- `all` (programmer): Remake all object modules, executables, etc. from files in the current directory.
- `allsys` (system): Remake all object modules, executables, etc. from files in the **AIPS++** directory.
- `clean` (programmer): Delete "rubbish" files in the current directory.
- `cleansys` (system): Delete "rubbish" files in the **AIPS++** directory.
- `chkout` (system): Update the **AIPS++** `code` directories by checking out files from the `rcs` directory as required.
- `makefile` (general): Update the makefile itself by checking it out of the `rcs` directory, or (for programmers) copying it from the **AIPS++** `code` directory.
- *class*`.i` (programmer): Preprocess class implementation file *class*`.C` (from the current directory if present) and leave it in the current directory.
- *class*`.o` (programmer): Recompile class implementation file *class*`.C` (from the current directory if present) and leave it in the current directory.
- *class*`TEST` (programmer): Recompile the class test program for *class*`.C` (from the current directory if present) and leave it in the current directory.
- `lib(`*class*`.o)` (system): Remake an object module *class*`.o` from *class*`.C` in the **AIPS++** directory and replace it in the library
- `lib` (system): Remake the object library associated with an `implement` directory.
- `ranlib` (system): *ranlib* the object library.
- *applic* (progammer): Recompile application *applic*`.C` and leave it in the current directory.
- `bin/`*applic* (system): Recompile application *applic*`.C` in the **AIPS++** directory and install it in the `bin` directory.

- `bin` (system): Recompile all applications in the **AIPS++** directory.
- `test_env` (general): Print out all relevant *make* variables defined within the makefile.
- `test_def` (general): Print out all relevant *make* variables defined in the `makedefs` files included by the makefile.

Library maintenance requires special attention to avoiding conflicts which may arise when two programmers independently try to update the library. This is handled by a script used by the makefiles called *updatelib*. Object modules are deposited by *make* in a library-specific subdirectory of `aips++/$ARCH/$VERS/tmp`, for example `...tmp/libkernel`. When all modules have been produced the makefile invokes *updatelib* which copies the library to the temporary area, updates it, *ranlib*s it, then copies it back to the `lib` area.

# 8 System databases

The `aipsrc` databases have been implemented via a `C`-program called *getrc*. It looks for device and other definitions in a format similar to that of the `.Xdefaults` database used by **X-windows**. In resolving a reference it searches the following sequence of `aipsrc` files:

```
~/.aipsrc
$AIPS/$ARCH/$VERS/$SITE/$HOST/aipsrc
$AIPS/$ARCH/$VERS/$SITE/aipsrc
$AIPS/$ARCH/$VERS/aipsrc
```

The last of these contains default values, and the other three allow these to be overridden on a user-, host-, and site-specific basis. A *-i* option provides for *getrc* to ignore the user's `aipsrc` file - it would not be appropriate for them to override access restrictions set by the local **AIPS++** administrator for example. A *-v* option causes *getrc* to indicate how it resolved the reference, and is useful for debugging.

The `aipsrc` mechanism works well and *getrc* is now used in a number of **AIPS++** scripts, including a simple and easily configurable set of printer utilities, and a utility for printing class header and implementation files in a compact and convenient form.

# 9 Future work

The following list indicates some areas of development/refinement in **AIPS++** system management. Some of the jobs listed could be done in under an hour, others would take several days.

- *exhale* should limit the number of incremental update files maintained in `~aips++/master_ftp` to about two-weeks worth.
- *exhale* should create a new *RCS* version when it creates a new base release.
- When *inhale* fetches a new base release from the `master` it should also update the `import_ftp` and `code_ftp` directories for redistribution to local end-users.
- Bug fixes should be stored in `~aips++/code_ftp` and a special-purpose script should be provided to fetch and apply them. *configure* needs to know that bug fixes must be unloaded *after* the base release files.
- As far as is possible, *configure* should check to see that entries made in the `makedefs` and `aipsrc` files are valid.
- Deletions from the `master` must be propagated to the `slave`.
- *updatelib* should test library modules (other than `Input.o`) to ensure they don't contain a `main()`.

- *ai* should test the code being checked in to ensure that it conforms to **AIPS++** coding style rules.
- All of the **AIPS++** shell scripts are written in Bourne shell. It may be advantageous in the long run to convert them to *perl* which is probably more portable. The exceptions are *aipsinit*, which must have Bourne and C-shell versions, and *configure* which can't assume that *perl* is available, and would have to install it if it wasn't.
- man pages are required for *ai*, `aipshosts`, *ao*, *au*, *configure*, `makedefs`, and *updatelib*.
- Makefile enhancements:
  - At the moment *make* will recompile everything if one header file has been changed because the makefiles don't contain a proper dependency analysis. Dependency information should be generated.
  - Separate optimized/nodebug (sharable) and nooptimize/debug (static) libraries should be maintained.
  - Library and application rebuilds should be structured in such a way that several machines can work together to rebuild the system.
  - The `makefile` target in all makefiles should copy from the **AIPS++** code directory if no `rcs` directory is present.
  - Directory search should be used for link libraries.
  - A makefile is required for the `~aips++/code/kernel/doc` areas.